ТРЕБОВАНИЯ К ПРОГРАММАМ

1. Программа работает с массивом объектов типа `data`:

```
# ifndef data_H
# define data_H
# include <stdio.h>

class data
{
  private:
    int m = M;
    double a[M] = {0};
    static int p;
  public:
    data () = default;
   ~data () = default;
    int get_m () const { return m; }
    const double  * get_a () { return a; }
    static int get_p () { return p; }
    static void set_p (int q)
      { p = q; }
    // Copy constructor
    data (const data &x) = default;
    // Assignement operator
    data& operator= (const data&) = default;
    // Comparison operators
    int operator< (const data& x) const
      { return m < x.m; }
    int operator> (const data& x) const
      { return m > x.m; }
    // Print data in a line (not more than p): m a[0] a[1] ... a[m-1]
    void print (FILE * fp = stdout);
    // Read data from the line: m a[0] a[1] ... a[m-1]
    int read (FILE *fp = stdin);
    // Init data by formulae with number s
    void init (int s);
};

# endif // data
```

   В класс можно добавлять и другие функции, если это требуется для решения задачи.

2. Параметр M указывается при компиляции всех файлов программы, например для $M = 10$:

```
g++ -DM=10 имя_файла.cpp
```

3. Программа должна получать все параметры в качестве аргументов командной строки.

4. Аргументы командной строки **для задач 4–10**:

   1) n – размерность массива,

2) p – количество выводимых значений в массиве,

3) s – задает номер формулы для инициализации массива, должен быть равен 0 при вводе массива из файла,

4) `filename` – имя файла, откуда надо прочитать массив. Этот аргумент **отсутствует**, если $s! = 0$.

Например, запуск

```
./a04.out 4 4 0 a.txt
```

означает, что массив длины 4 надо прочитать из файла `a.txt`, и выводить не более 4-х элементов массива, а запуск

```
./a04.out 1000000 6 1
```

означает, что массив длины 1000000 надо инициализировать по формуле номер 1, и выводить не более 6-ти элементов массива.

5. Аргументы командной строки **для задачи 1**: добавляется **дополнительный первый аргумент**, остальные аргументы – как в задачах 4–10:

   1) x – дополнительный аргумент (x – имя файла, откуда надо прочитать один объект типа `data`)

   2) n – размерность массива,

   3) p – количество выводимых значений в массиве,

   4) s – задает номер формулы для инициализации массива, должен быть равен 0 при вводе массива из файла,

   5) `filename` – имя файла, откуда надо прочитать массив. Этот аргумент **отсутствует**, если $s! = 0$.

6. Аргументы командной строки **для задачи 3**: добавляется **дополнительный первый аргумент**, остальные аргументы – как в задачах 4–10:

   1) m – дополнительный аргумент (m целое число – индекс элемента в массиве ($m = 0, \ldots, n-1$)),

   2) n – размерность массива,

   3) p – количество выводимых значений в массиве,

   4) s – задает номер формулы для инициализации массива, должен быть равен 0 при вводе массива из файла,

   5) `filename` – имя файла, откуда надо прочитать массив. Этот аргумент **отсутствует**, если $s! = 0$.

7. Аргументы командной строки **для задачи 2**: для задания двух входных массивов используется удвоенный "комплект" аргументов из задач 4–10:

   1) n – размерность массива a,

   2) p_a – количество выводимых значений в массиве a,

   3) s_a – задает номер формулы для инициализации массива a, должен быть равен 0 при вводе массива из файла,

   4) $\texttt{filename}_a$ – имя файла, откуда надо прочитать массив. Этот аргумент **отсутствует**, если $s_a! = 0$,

5) m – размерность массива b,

6) p_b – количество выводимых значений в массиве b,

7) s_b – задает номер формулы для инициализации массива b, должен быть равен 0 при вводе массива из файла,

8) `filename`$_b$ – имя файла, откуда надо прочитать массив. Этот аргумент **отсутствует**, если $s_b! = 0$.

Например, запуск

```
./a02.out 4 4 0 a.txt 6 6 0 b.txt
```

означает, что массив a длины 4 надо прочитать из файла `a.txt`, выводить не более 4-х элементов массива, массив b длины 6 надо прочитать из файла `b.txt`, выводить не более 6-ти элементов массива, запуск

```
./a02.out 1000000 6 1 8 6 0 b.txt
```

означает, что массив a длины 1000000 надо инициализировать по формуле номер 1, выводить не более 6-ти элементов массива, массив b длины 8 надо прочитать из файла `b.txt`, выводить не более 6-ти элементов массива, запуск

```
./a02.out 1000000 6 1 2000000 6 5
```

означает, что массив a длины 1000000 надо инициализировать по формуле номер 1, выводить не более 6-ти элементов массива, массив b длины 2000000 надо инициализировать по формуле номер 5, выводить не более 6-ти элементов массива.

8. Ввод массива должен быть оформлен в виде подпрограммы, находящейся в отдельном файле.

9. Ввод массива из файла. В указанном файле находится массив в формате:

$$\begin{array}{cccccc} m_1 & a[0] & a[1] & \dots & a[m_1-1] \\ m_2 & a[0] & a[1] & \dots & a[m_2-1] \\ \dots & \dots & \dots & \ddots & \dots \\ m_n & a[0] & a[1] & \dots & a[m_n-1] \end{array}$$

где n - указанный размер массива. Если $m_i > M$, то полагается $m_i = M$ и считывается M элементов строки, остальные элементы до конца строки пропускаются. Программа должна выводить сообщение об ошибке, если указанный файл не может быть прочитан, содержит меньше n элементов массива или данные неверного формата.

10. Ввод массива по формуле. Элемент a_i массива A полагается равным

$$a_i = \{m = f(s,n,i), \quad a[j] = 0, j = 0,1,\dots,m-1\}, \quad i = 1,\dots,n, \tag{1}$$

где $f(s,n,i)$ - функция, которая возвращает значение для поля n (i)-го элемента массива по формуле номер s (аргумент командной строки). Функция $f(s,n,i)$ должна быть оформлена в виде отдельной подпрограммы `int f (int s, int n, int i);` работающей только с целыми числами.

$$f(s,n,i) = \begin{cases} i & \text{при} \quad s = 1 \\ n-i & \text{при} \quad s = 2 \\ i/2 & \text{при} \quad s = 3 \\ n-i/2 & \text{при} \quad s = 4 \end{cases}$$

Инициализация объекта по формуле (1) должна быть оформлена как член класса `data`.

11. Решение задачи должно быть оформлено в виде подпрограммы, находящейся в отдельном файле и получающей в качестве аргументов массив и его длину (в 1 – 3 задачах также дополнительные аргументы). Получать в этой подпрограмме дополнительную информацию извне через глобальные переменные, включаемые файлы и т.п. запрещается.

12. Программа должна содержать подпрограмму вывода на экран массива длины не более p. Эта подпрограмма используется для вывода исходного массива после его инициализации, а также для вывода на экран результата. Подпрограмма выводит на экран не более, чем p элементов массива, где p – параметр этой подпрограммы (аргумент командной строки). Каждый элемент массива должен печататься на новой строке.

13. Программа должна содержать подпрограмму для вычисления количества элементов массива, меньших предыдущего элемента.

14. Вывод результата работы функции в функции `main` должен производиться по формату:

```
printf ("%s : Task = %d Diff = %d Elapsed = %.2f\n",
        argv[0], task, diff, t);
```

где

- `argv[0]` – первый аргумент командной строки (имя образа программы),
- `task` – номер задачи (1–10),
- `diff` – возвращаемое значение функции, вычисляющей количество элементов массива, меньших предыдущего элемента, **после** работы функции, реализующей решение этой задачи,
- `t` – время работы функции, реализующей решение этой задачи.

Вывод должен производиться в точности в таком формате, чтобы можно было автоматизировать обработку запуска многих тестов.

Задачи

1. Написать функцию, получающую в качестве аргументов неубывающий массив $a[n]$ объектов типа `data`, целое число n, являющееся длиной этого массива и ссылку на объект типа `data` x, и возвращающую место, где в этот массив можно вставить x (т.е. целое число i такое, что $a[i] \le x \le a[i+1]$). Место определяется двоичным поиском по следующему алгоритму (*алгоритм деления пополам*).

   Взять первоначально 0 и n в качестве границ поиска элемента; далее, до тех пор, пока границы не совпадут, шаг за шагом сдвигать эти границы следующим образом: сравнить x с $a[s]$, где s – целая часть среднего арифметического границ; если $a[s] < x$, то заменить прежнюю нижнюю границу на $s+1$, а верхнюю оставить без изменений, иначе оставить без изменения нижнюю границу, а верхнюю заменить на s; когда границы совпадут, став равными некоторому числу t, выполнение закончится с результатом t.

   Общее количество сравнений и перестановок элементов в наихудшем случае не должно превышать $\log_2 n + O(1)$. Основная программа должна заполнять данными массив, выводить его на экран, вводить с командной строки x, вызывать эту функцию и выводить на экран результат ее работы.

2. Написать подпрограмму, получающую в качестве аргументов три массива $a[n]$, $b[m]$, $c[n+m]$ объектов типа `data`, где $a[n]$, $b[m]$ не убывают, и целые числа n и m, и строящую по неубывающим массивам $a[n]$, $b[m]$ неубывающий массив $c[n+m]$ слиянием первых двух за $n+m+O(1)$ сравнений и $n+m+O(1)$ пересылок элементов по следующему алгоритму

Просматриваем очередные элементы $a[i]$, $i=0,\ldots,n-1$ и $b[j]$, $j=0,\ldots,m-1$ массивов a и b. Если $a[i]<b[j]$, то $c[k]=a[i]$ и увеличиваем i на 1, иначе $c[k]=b[j]$ и увеличиваем j на 1. Здесь k, $k=0,\ldots,n+m-1$ – очередной элемент массива c (здесь всегда равен $i+j$).

Основная программа должна заполнять данными массивы $a[n]$ и $b[m]$, выводить их на экран, вызывать эту подпрограмму и выводить на экран результат ее работы – массив $c[n+m]$.

3. Написать функцию, получающую в качестве аргументов массив $a[n]$ объектов типа `data`, целое число n, являющееся длиной этого массива, целое число m – индекс элемента в массиве ($m=0,\ldots,n-1$), и переставляющую элементы массива так, чтобы вначале шли все элементы, не большие $a[m]$, а затем элементы, не меньшие $a[m]$. Функция возвращает новую позицию элемента $a[m]$ в результирующем массиве (если элемент со значением $a[m]$ не единственный, то новую позицию любого равного ему элемента). Решение задачи производится следующим алгоритмом:

   Просматривая массив с начала (линейным поиском), находим i такое, что $a[i]\geq a[m]$. Просматривая массив с конца (линейным поиском), находим j такое, что $a[j]\leq a[m]$. Если $i\leq j$, то меняем $a[i]$ и $a[j]$ местами, если оказалось, что m равно i или j, то обновляем m, затем i увеличиваем на 1, j уменьшаем на 1. Повторяем эту процедуру пока $i\leq j$. После этого все элементы $a[0],\ldots,a[i]$ будут не больше всех элементов $a[j],\ldots,a[n-1]$. Возвращаемым значением функции будет $i=j$.

   Общее количество сравнений элементов в наихудшем случае не должно превышать $n/2+O(1)$, а пересылок элементов – $3n/2+O(1)$. Основная программа должна заполнять данными массив, выводить его на экран, вводить с командной строки x, вызывать эту функцию и выводить на экран результат ее работы: массив $a[n]$ и значение $i=j$.

4. Написать подпрограмму, получающую в качестве аргументов массив $a[n]$ объектов типа `data` и целое число n, и переставляющую элементы массива $a[n]$ в неубывающем порядке: $a[0]\leq a[1]\leq\cdots\leq a[n-1]$ по следующему алгоритму (*"пузырьковая"сортировка*):

   Последовательным просмотром $a[0],a[1],\ldots,a[n-1]$ найти наименьшее i такое, что $a[i]>a[i+1]$. Поменять $a[i]$ и $a[i+1]$ местами, возобновить просмотр с элемента $a[i+1]$ и т.д. Тем самым наибольший элемент передвинется на последнее место. Следующие просмотры начинать опять сначала, уменьшая на 1 количество просматриваемых элементов. Массив будет упорядочен после просмотра, в котором участвовали только первый и второй элементы.

   Общее количество сравнений в наихудшем случае не должно превышать $n^2/2+O(n)$, а пересылок элементов – $3n^2/2+O(n)$. Основная программа должна заполнять данными массив, выводить его на экран, вызывать эту подпрограмму и выводить на экран результат ее работы.

5. Написать подпрограмму, получающую в качестве аргументов массив $a[n]$ объектов типа `data` и целое число n, и переставляющую элементы массива $a[n]$ в неубывающем порядке: $a[0]\leq a[1]\leq\cdots\leq a[n-1]$ по следующему алгоритму (*сортировка нахождением минимума*):

   Найти элемент массива, имеющий наименьшее значение, переставить его с первым элементом, затем проделать то же самое, начав со второго элемента и т.д.

   Общее количество сравнений в наихудшем случае не должно превышать $n^2/2+O(n)$, а пересылок элементов – $3n+O(1)$. Основная программа должна заполнять данными массив, выводить его на экран, вызывать эту подпрограмму и выводить на экран результат ее работы.

6. Написать подпрограмму, получающую в качестве аргументов массив $a[n]$ объектов типа `data` и целое число n, и переставляющую элементы массива $a[n]$ в неубывающем порядке: $a[0]\leq a[1]\leq\cdots\leq a[n-1]$ по следующему алгоритму (*сортировка линейной вставкой*):

   Просматривать последовательно $a[1],\ldots,a[n-1]$ и каждый новый элемент $a[i]$ вставлять на подходящее место в уже упорядоченную совокупность $a[0],\ldots,a[i-1]$. Это место определяется последовательным сравнением $a[i]$ с упорядоченными элементами $a[0],\ldots,a[i-1]$.

Общее количество сравнений и пересылок элементов в наихудшем случае не должно превышать $n^2/2+O(n)$ (каждое). Основная программа должна заполнять данными массив, выводить его на экран, вызывать эту подпрограмму и выводить на экран результат ее работы.

7. Написать подпрограмму, получающую в качестве аргументов массив $a[n]$ объектов типа `data` и целое число n, и переставляющую элементы массива $a[n]$ в неубывающем порядке: $a[0] \le a[1] \le \cdots \le a[n-1]$ по следующему алгоритму (*сортировка двоичной вставкой*):

   Просматривать последовательно $a[1],\ldots,a[n-1]$ и каждый новый элемент $a[i]$ вставлять на подходящее место в уже упорядоченную совокупность $a[0],\ldots,a[i-1]$. Это место определяется алгоритмом деления пополам (см. задачу 1).

   Общее количество сравнений в наихудшем случае не должно превышать $n\log_2 n+O(n)$, а пересылок элементов – $n^2/2+O(n)$. Основная программа должна заполнять данными массив, выводить его на экран, вызывать эту подпрограмму и выводить на экран результат ее работы.

8. Написать подпрограмму, получающую в качестве аргументов массив $a[n]$ и вспомогательный массив $b[n]$ объектов типа `data`, и целое число n, и переставляющую элементы массива $a[n]$ в неубывающем порядке: $a[0] \le a[1] \le \cdots \le a[n-1]$ по следующему алгоритму (*сортировка Неймана*):

   Вначале весь массив рассматривается как совокупность упорядоченных групп по одному элементу в каждом. Слиянием соседних групп (см. задачу 2) получаются упорядоченные группы, каждая из которых содержит два элемента (кроме, может быть, последней группы, которой не нашлось парной). Далее упорядоченные группы укрупняются тем же способом и т.д. Используется вспомогательный массив b[n].

   Общее количество сравнений и пересылок элементов в наихудшем случае не должно превышать $n\log_2 n+O(n)$ (каждое). Основная программа должна заполнять данными массив, выводить его на экран, вызывать эту подпрограмму и выводить на экран результат ее работы.

9. Написать подпрограмму, получающую в качестве аргументов массив $a[n]$ объектов типа `data` и целое число n, и переставляющую элементы массива $a[n]$ в неубывающем порядке: $a[0] \le a[1] \le \cdots \le a[n-1]$ по следующему алгоритму (*"быстрая"сортировка*):

   Полагаем $x=a[n/2]$. Алгоритмом задачи 3 находим место, где в этот массив можно вставить x, т.е. такое i, что все элементы $a[0],\ldots,a[i-1]$ будут меньше или равны всех элементов $a[i],\ldots,a[n-1]$. Затем сортируем каждую из частей (т.е. $a[0],\ldots,a[i-1]$ и $a[i],\ldots,a[n-1]$) этим же алгоритмом. Для уменьшения глубины рекурсии вначале сортируем более короткий массив. Затем вместо повторного вызова процедуры переходим к ее началу с новыми значениями указателя на начало массива a и его длины n.

   Общее количество сравнений в наихудшем случае не должно превышать $n^2/2+O(n)$, а пересылок элементов – $3n^2/2+O(n)$. Основная программа должна заполнять данными массив, выводить его на экран, вызывать эту подпрограмму и выводить на экран результат ее работы.

10. Написать подпрограмму, получающую в качестве аргументов массив $a[n]$ объектов типа `data` и целое число n, и переставляющую элементы массива $a[n]$ в неубывающем порядке: $a[0] \le a[1] \le \cdots \le a[n-1]$ по следующему алгоритму (*"турнирная"сортировка или алгоритм heapsort*):

    Массив $a[n]$ рассматриваем как бинарное дерево с корнем $a[0]$. Для элемента с номером $a[k]$ потомками являются элементы $a[2*k+1]$ и $a[2*k+2]$, а родителем – элемент $a[(k-1)/2]$. На первом этапе алгоритма превращаем наше бинарное дерево в т.н. упорядоченную пирамиду, т.е. в дерево, в котором всякая цепочка от корня до любого конечного элемента выстроена по убыванию. Для этого для всех $k=1,\ldots,n-1$ идем от элемента $a[k]$ по цепочке его родителей, продвигая его на свое место (подобно тому, как это делалось в алгоритме сортировки линейной вставкой). После этого этапа алгоритма в корне дерева (т.е. в $a[0]$) будет находиться максимальный элемент массива.

На втором этапе алгоритма для всех $k = n-1, \ldots, 1$: меняем корень (т.е. $a[0]$) и элемент $a[k]$ местами и рассматриваем массив a как имеющий длину k. В этом массиве восстанавливаем структуру упорядоченной пирамиды, нарушенную помещением элемента $a[k]$ в $a[0]$. Для этого идем от элемента $a[0]$ по цепочкам его потомков, продвигая его на свое место (подобно тому, как это делалось в алгоритме сортировки линейной вставкой, с одним отличием: элемент пирамиды должен быть больше обоих своих потомков).

Общее количество сравнений в наихудшем случае не должно превышать $4n\log_2 n + O(n)$, а пересылок элементов – $2n\log_2 n + O(n)$. Основная программа должна заполнять данными массив, выводить его на экран, вызывать эту подпрограмму и выводить на экран результат ее работы.